Network Traffic Based Detection of Repackaged Android Apps via Mobile Fog Computing

***MA Rahim Khan**[1]
[*]Khan_rahim@rediffmail.com

Dr. Shivani Dubey[2]
shivanidubey@lingayasvidyapeeth.edu.in

Dr. RC Tripathi[3]
rctripathig@gmail.com
[123]Department of Computer Science and Engineering Lingaya's Vidyapeeth, Faridabad, Haryana India
[1]College of Computer and Information Sciences Majmaah University Almajmaah,11952 Saudi Arabia

Abstract
Repackaged apps are the biggest challenge in the Android ecosystem. Many tools are available in the market to detect the repackaged apps. Most of the tools and recent researches comparing the original apps with suspected apps to detect the repackaged apps. Into this one more biggest challenge to choose the original. The solution to this problem to make an official android store for all original apps. However, the pairwise comparison is not the optimum solution because it is time-consuming and inefficient to repackage detection apps.

This paper proposes an approach to detect the repackaged apps using network traffic-based clustering using Fog Computing. The main observation to collect the network traffics to trace the plaintext for original and repackaged app detection. During analyzing of traffics, extracting the statistical feature, calculate the similarity between app. Here, the clustering automatically separates both types of apps, such as a repackaged and original app. The experiment analyses the performance of the proposed approach with minimum complexity and high precision and reduces the effort of mobile computing to resolve the malware.

Keywords: Network Traffic, Cluster analysis, Repackage app, feature Extraction, Mobile Fog Computing, Android Malware

1. INTRODUCTION

Presently, Android devices (tablet, smartwatch, and smartphone) have speedy popularity worldwide. As per one website survey of Statista, The Android mobile OS is counted more than 73.48% on Dec 2020 in the Global market [1]. The leading cause of the popularity of Android OS, it is open source and conveniently downloads android apps from various play store. The most famous app store is the Google play store, which has approx. Three million apps by Feb 2021 [2]. These feature-rich apps types make the Android ecosystem more attractive and vibrant.

Android app has the extension *.APK (Android Package Kit) in ZIP format, including Share Object(SO), manifest, class.DEX. Developer byte code. Android apps are easily de-assembled or repackaged by some popular tools such as apktool1 and Jadx2 , which can quickly add malicious code or add some extra ad's ware. The malicious code is used to steal sensitive information, and ad code is used to earn financial gain. Therefore, repackaging app has become a significant threat to the android ecosystem [3].

In [4-6], apps embedded with malicious code, detect as repackaged apps. The majority of malware are repackaged apps. Authors observed that repackaging becomes a substantial threat to secure Android environments. Several approaches have been proposed to investigate the detection of repackaged apps [7-14]. Previous work has two categories to detect malware, such as off-device and In-device. Off-devices dynamic run the APK in the virtual environment. It extracts the signature of APK and records the behaviors of apps. If any malicious code or unpredictable behaviors are diagnosed, treat it as malware. Those approaches are designed as network traffic-based detection of malware.

On-device detection, These techniques judged the app's behavior on local user devices; it recognizes the malware during the installation process of APK. If we compare between off-device and the On-device approach, the on-device approach has limited space, resources to detect the malware. However, in off-device approach have extensive resources, power, and space. Nevertheless, on-device is more challenging than off-devices due to limited resources.

Several approaches have been proposed to detect malware on mobile devices. The first approach to observing the operating system behavior, API calls, and application network behavior should be extracted and defined—the subsequent detection of malware based on feature extraction. Current approaches are divided into client and server-side detection; client-side approaches detect the malware on mobile devices, and server-side approaches detect the malware into apps on the remote server. In practice, client-side approaches have restrictions due to limited resources and power to detect malware into apps [8]. However, server-side approaches have extensive resources, more power, and high performance [15]. The reason, detection feature is collected and processed on mobile devices.

This paper proposed an approach to detect the repackage malware based on clustering via Mobile Fog Computing (MFC). The main strategy of our method to extract the feature of apps and pre-evaluate offload to Fog. Therefore, the workload of mobile devices is reduced. Our proposed architecture has three components to detect Android Malware: Mobile devices, Fog Server, and cloud. The Fog server extracts the detection feature from mobile network traffic; this approach does preprocess, which helps to reduce the workload of mobile devices, minimize the amount of data to send, and protect mobile users' privacy. The extracted feature from network traffics are sent to the cloud for result processing. The app's similarity is evaluated based on extractive features on the cloud, deciding the app is the original app or repackaged malware. The based on similarities of the app, automatically clustering separated the original app and repackaged malware. We designed the comprehensive set of experiments, found an average accuracy rate of 97.7%.

Our main contribution in proposed works is significantly less complex and effective frameworks based on fog computing for detecting repackaged malware. This is the server-side approach, which is reduced the workload of mobile devices. In addition, it has network traffics-based feature extraction; it has a reasonable accuracy rate of detection and protection of mobile user’s privacy—the identification or detection of repackaged Android malware done through the network traffic clustering approach.

We cluster the network traffics that generated the app to conclude which app has been repackaged.

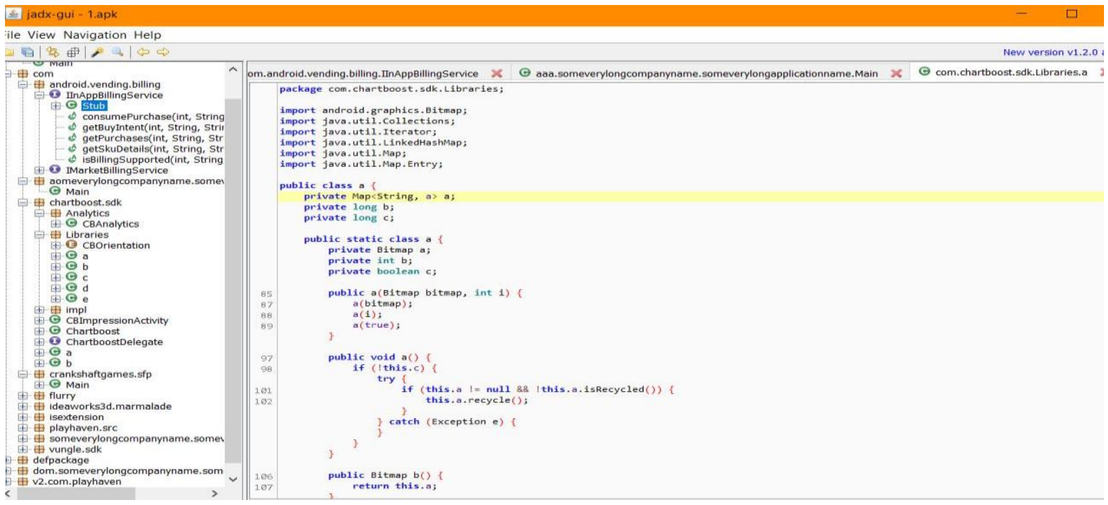
A proposed approach for network traffic clustering to detect the repackaged android malware. Here, the main aim to generate network traffic by the same app to determine that the app has been repackaged or not. Primary, we determine the similarities between app by statistical flow feature and plaintext contents. Then, automatically similarities rate is clustering to separate the benign and repacked app. Some models extract from network behaviors the signature matching process in advance for malware detection; it is not deal with encryption and zero-day attack. So our approach more efficient and less complex in practice.

2. PRELIMINARIES

We are introducing Android malware and Fog Computing from the primary concept of detecting the repackaged app.

2.1 Repackaged App

Android Apk is very easy to repackage. As shown in Figure 1(a),1(b), the 1.APK was dissembled by reverse engineering tools JADX. The JADX is the power tool for reverse engineering.


jadx-gui - 1.apk
File View Navigation Help
New version v1.2.0
com
android.vending.billing
IInAppBillingService
Stub
consumePurchase(int, String
getBuyIntent(int, String, Stri
getPurchases(int, String, Str
getSkuDetails(int, String, Str
isBillingSupported(int, String
IMarketBillingService
aomeverylongcompanyname.somev
Main
chartboost.sdk
Analytics
CBAnalytics
Libraries
CBOrientation
a
b
c
d
e
impl
CBImpressionActivity
Chartboost
ChartboostDelegate
a
b
crankshaftgames.sfp
Main
flurry
ideaworks3d.marmalade
isextension
playhaven.src
someverylongcompanyname.somev
vungle.sdk
defpackage
dom.someverylongcompanyname.som
v2.com.playhaven
om.android.vending.billing.IInAppBillingService
aaa.someverylongcompanyname.someverylongapplicationname.Main
com.chartboost.sdk.Libraries.a

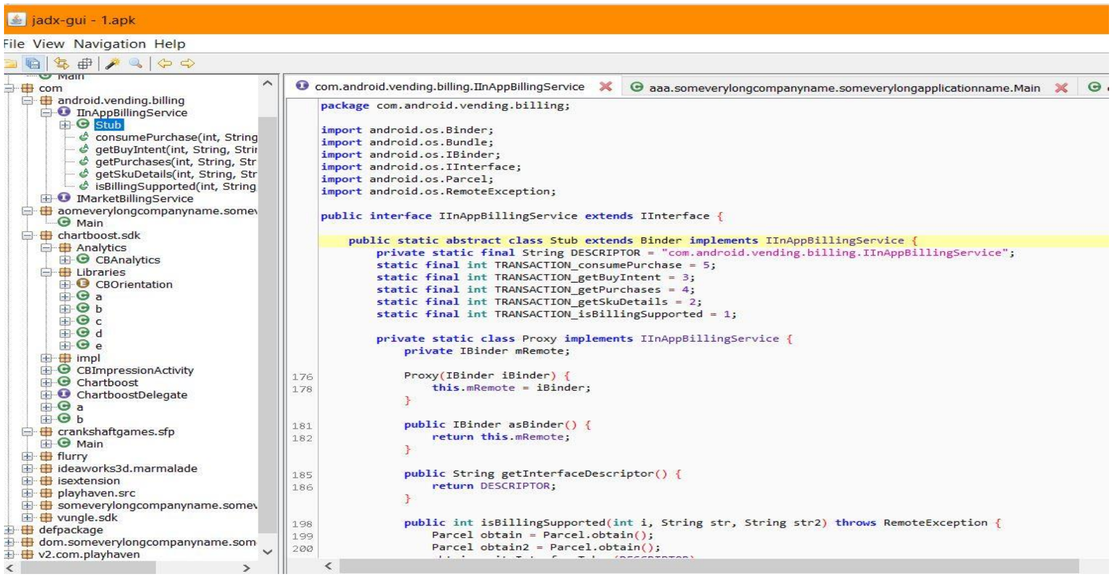
```
package com.chartboost.sdk.Libraries;

import android.graphics.Bitmap;
import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Map.Entry;

public class a {
    private Map<String, a> a;
    private long b;
    private long c;

    public static class a {
        private Bitmap a;
        private int b;
        private boolean c;

        public a(Bitmap bitmap, int i) {
            a(bitmap);
            a(i);
            a(true);
        }

        public void a() {
            if (!this.c) {
                try {
                    if (this.a != null && !this.a.isRecycled()) {
                        this.a.recycle();
                    }
                } catch (Exception e) {
                }
            }
        }

        public Bitmap b() {
            return this.a;
```

Figure 1(a): Reverse Engineering by JADX tool, Total File structure of APK and related code

Most Android malware generates repackaged apps from benign apps; the most popular app is embedded in the malicious code [4]. As proof [16], a community of android security shown 80% of apps are repacked app—the repackaged app launched on the third-party app store. Third-party app stores did not require any integrity evolution of uploaded app [17]. In F-Droid, found some of the app repackaged by the same developer [18]. The Repackaged app is remarked by evaluating by same network traffic app [7,14].

An Android app always has a remote connection to the server to interact, Android app request command to the server, and receives sensitive data from the server. If there is malicious code in the app, network behavior will be differing from the benign app.

Therefore, benign apps lot of network traffic differences from repackaged apps. Inspired by this kind of comparison, a proposed an approach based on network traffic clustering to separate the benign app and repackaged app.


jadx-gui - 1.apk
File View Navigation Help
com
android.vending.billing
IInAppBillingService
Stub
consumePurchase(int, String
getBuyIntent(int, String, Stri
getPurchases(int, String, Str
getSkuDetails(int, String, Str
isBillingSupported(int, String
IMarketBillingService
aomeverylongcompanyname.somev
Main
chartboost.sdk
Analytics
CBAnalytics
Libraries
CBOrientation
a
b
c
d
e
impl
CBImpressionActivity
Chartboost
ChartboostDelegate
a
b
crankshaftgames.sfp
Main
flurry
ideaworks3d.marmalade
isextension
playhaven.src
someverylongcompanyname.somev
vungle.sdk
defpackage
dom.someverylongcompanyname.som
v2.com.playhaven
com.android.vending.billing.IInAppBillingService
aaa.someverylongcompanyname.someverylongapplicationname.Main

```
package com.android.vending.billing;

import android.os.Binder;
import android.os.Bundle;
import android.os.IBinder;
import android.os.IInterface;
import android.os.Parcel;
import android.os.RemoteException;

public interface IInAppBillingService extends IInterface {

    public static abstract class Stub extends Binder implements IInAppBillingService {
        private static final String DESCRIPTOR = "com.android.vending.billing.IInAppBillingService";
        static final int TRANSACTION_consumePurchase = 5;
        static final int TRANSACTION_getBuyIntent = 3;
        static final int TRANSACTION_getPurchases = 4;
        static final int TRANSACTION_getSkuDetails = 2;
        static final int TRANSACTION_isBillingSupported = 1;

        private static class Proxy implements IInAppBillingService {
            private IBinder mRemote;

            Proxy(IBinder iBinder) {
                this.mRemote = iBinder;
            }

            public IBinder asBinder() {
                return this.mRemote;
            }

            public String getInterfaceDescriptor() {
                return DESCRIPTOR;
            }

            public int isBillingSupported(int i, String str, String str2) throws RemoteException {
                Parcel obtain = Parcel.obtain();
                Parcel obtain2 = Parcel.obtain();
```

Figure 1(b) Reverse Engineering of Com. Android. vendling.Billing.AppbillingServices

2.2 Fog Computing

Fog computing is technology, which uses the edge of the network on downstream data on behalf of the Internet of Things (IoT) amenities [19]. The important thing in Fog computing, it has the edge, consist computation and network resources between mobile-device/IoT and cloud. Figure 2 represents the paradigm of edge computing. Edges network deployed by Fog server. It is used for preprocessing and foreword the extracted feature of network traffic to the cloud.

The transfer of the service on fog server, fog server is updated online. In most scenarios, the device is connected to the Fog server and cloud simultaneously. In this works, it represents a solid and dotted line. Because in this scenario, all data are not preprocessed by the fog server. The fog computing paradigm represents in Figure 2 for Android malware detection.

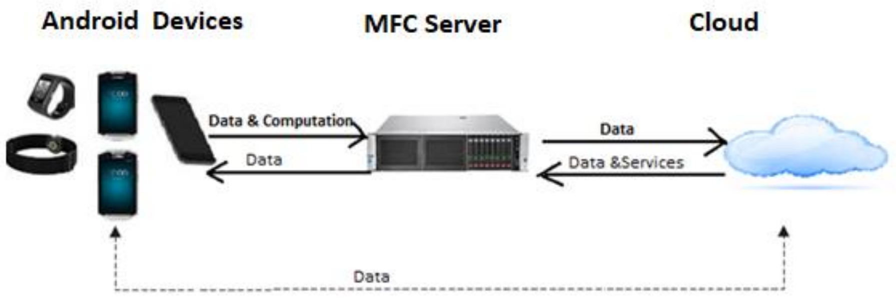


Figure 2: Paradigm of Fog computing

3. RELATED WORK:

The repackaged Android app is the biggest source of malware; many approaches have been proposed to detect them. Similarly, DroidMoss and ResDroid proposed an approach to detecting repackaged apps in the Android Market [20,21]. DroidMoss focused on code and pairwise similarity comparison. ResDroid extracted structure feature and event handler information from code and core resources. Later on, the extracted feature was used to detect the repackaged android app. It used the dividend- conquer strategy to reduce the computation time and have a parallel processing approach.

Assuming Google play store were benign apps, Massvet [22] downloaded all the apps in advance and constructed the V-Score and M-Score database to detect the repackaged android app. In addition, code similarities comparison, behavior, and UI-based approaches are more emerging—DroidChain [23] composed the typical behaviors of the app to detect the repackaged malware. In DroidChain, models composed of typical behavior have some issues, due to this leakage the privacy, escalating privilege, malware installation.

According to an Approach [24], it cannot detect the repackaged partly due to their behavioral similarities of original apps. To overcome this issue, I designed a new technique based on code heterogeneity analysis to detect repackaged Android malware. The hole app structure is partitioned into multiple dependent regions, each region independently classified on its behavioral features.

Techniques used the VizMal[25] for visualizing the app execution to trace to detect the repackaged Android malware and noticed the expected malicious behaviors. A proposed approach used the static UI feature [26]; it detected 3723 and 15956 repackaged apps in the Anzhi market and MI Market, respectively. Results have shown that repackaged malware is a severe threat to the security of mobile. Even the Android market [27] uses the server approached to detect the repackaged app.

Presently, several approaches are proposed to detect the repackaged Android malware based on network traffic behaviors. Arora et al. proposed an approach based on usefulness network traffic to detect Android Malware [28]. One more technique based on network traffic divided app HTTP into two categories: primary and Nonprimary modules. Primary modules used the HTTP flow distance algorithm and the Hungarian method to determine the traffic and pairs similarity. CREDROID [29] proposed an approach to detect the malicious app based on domain name system(DNS) requests and data transmitted to remote servers by performing the in-depth analysis of network traffic logs in offline mode.

Zulkifli et al. [30] designed an approach to detect the repackaged malware app, which used seven network traffic features with the J48 decision tree algorithm. This proposed approach [31] to detect the repackaged Android malware app based on comparing network behaviors of similar apps. This paper used lightweight mobile edge computing. In this worked not consider repackaged apps in our knowledge. Our worked-based network traffic-based repackaged Android malware app detection. In our work, consider the network traffic only. In previous work such as [6,12], we do not need the compare each evaluated app to all apps in the Android store. We used MFC sever to collect network traffic of apps to determine the repackaged Android malware. In the process, we compare only the network traffic set. The statistical flow feature of the app also helps to deal with encrypted network traffic.

Father improves the performance of the proposed approach; Crowdroid [10] sends the detected feature to on central server for processing. This approach recoded the system call for processing to detect its feature; it is a very lightweight process. The Proposed methods used the permission base malware detection approach. This method divided the system into three categories: Signature Database, Android Client, and Central Server. The android app delivers to the central server by the client Android. The central server extracts all permission requests, calculates permission malware score etc.

Table 1: Download Malicious APK from ANDOZOO with size and Detection from TotalVirus.com

S.N.	APK	Date	Size MB	Score Detection by engines
1	939b53399a59b34d839dc055682db549.APK	2019-04-02	53.02	6/59
2	092b3663cb136f5192bfa7e1dc147420.APK	2019-08-17	93.02	5/59
3	093db22b159645d0d7201d9048f0541b.APK	2019-07-25	197.56	11/60
4	194adfa70978e2828a9397204b33bc51.APK	2019-05-27	120.47	3/60
5	220c905d9f1286285356c9511be9d581.APK	2019-05-20	93.80	12/61
6	271e7feb283bd249755be37c7ebf717a.APK	2019-04-01	98.41	14/57
7	293d5e8a0faf770cd00367e6e84ce2a8.APK	2018-09-07	118.26	23/62
8	349f4cf8ce6ab7652da35a89aaa10b3e.APK	2020-05-18	93.87	0/63
9	508e82e3ec3944ce257f126363d4ea7d.APK	2019-07-30	234.41	9/59
10	698b81d63841362c35be0543870b4a5d.APK	2021-02-05	121.37	8/61
11	718e60ac1ad4bd6e9e3ee46994ac8d68.APK	2019-05-17	127.33	6/59
12	814e01e693bea00db481dc30be5aeb1f.APK	2018-09-01	138.87	25/62
13	916a1677ec6fd9798a82381d921a5b9b.APK	2019-06-23	100.26	16/62
14	1317f0080e1b5d0a55876116e1dfd22b.APK	2019-08-04	108.43	3/59
15	642ec177fa7db0bb206b1fd2746798db.APK	2019-07-13	129.24	0/51
16	98f4b2087b408ee174fd07f7898b9127e.APK	2020-03-26	6.81	0/58
17	87b179e18c1947e365a7c9db4f0a3052.APK	2019-05-11	85.85	6/61
18	87b6994c27f474b9874f292ee27bda76.APK	2019-04-02	91.84	4/57
19	408f5eca5cd8be5623fe2fcbfb84d093.APK	2018-08-30	91.61	1/61
20	986ad626e433d8c7f9533fd33540ef1b.APK	2020-05-18	82.76	5/58

As Table 1 Reports, all engines cannot find all kinds of malicious apps; some engines can catch the malicious app. Even some apps did not catch by a single-engine.

4. MOTIVATION:

Most popular apps are chosen for repackaged apps and spread across the world. Therefore, existing techniques compare the vetted app with the official android market app such as google play store to distinguish the repackaged app. The main hypothesis of google play has benign and original apps. Official android app market deployed extensive security measures to protect from attackers. Every day, Google playtest the security of millions of apps, after that appeared in the play store. Awkwardly, there is a comparison between different apps, which is not appropriate as per complexity. Assume that we have N suspected malicious apps and M number official Android market apps. It requires N*M comparisons to detect the malicious app. Google play store has approximately 3 million apps. So that, offline comparison requires downloading millions of apps; it is a time-consuming process. Therefore, existing approaches have very high complexity and very difficult to apply in practice.

Our proposed approach has a special kind of network traffic management used to detect the repackaged malware apps. Figure 4 shown the MFC Server to manage specific user requests directly at network edges. Fog Server manages network traffic and forewords only specific information to the cloud instead of all remote traffic. Our main aim here to reduce the number of comparisons, same app run of different Android devices to produce network traffics for analyses and compared.

In proposed approach doing the traffic labeling and identifying app by using network traffic. Meanwhile, due to changes in a resource file or added malicious code, partially network traffic would be different from the original app network traffic. A repackaged app can easily identify. That is shown in Figure 5.

5. METHODOLOGY

Every device containing the android OS has limited resources; therefore, the detection of Android malware on the device has a very challenging task. We proposed an approach to detect the repackaged malware App at run time. The proposed framework of detection of repackaged Android App is illustrated in figure 4. Our frameworks are divided into three layers such Android devices, MFC Server, and Cloud layer. Figure 4, i repackaged apps install on u different android devices. MFC client installed on android devices to foreword the network traffic to MFC server for preprocessing. MFC server primarily labels the network traffics generated by i apps, u traffic set are captured. At that point, u traffic set filtered and get rid of the expected traffic from the set. In Figure 4 represent the labeling and filtering in step 1-3, traffic behaviors and similarity calculation in step 4-5, In step applying clustering approach to separate the repackaged app and original app—the technical detail of the steps discussed in the next section.

5.1 Traffic Labeling, filtering, and Feature extraction:

Traffic labeling planned to recognized all traffic produced by the app i for every android device. We have Extensive approaches to identify the app and flow of network traffics [32]. In MFC Server, traffic labeling is directed by accumulating HTTP signature[33] and correlation [34] of traffics. Every HTTP header has a unique key-value pair in the HTTP signature, and every app has a unique set shown in figure 6. In proposed approaches found that 90% of apps identify from HTTP signature. While HTTP signature matching could not identify by in encrypted apps, further identify the unknown traffic using the traffic correlation methods.

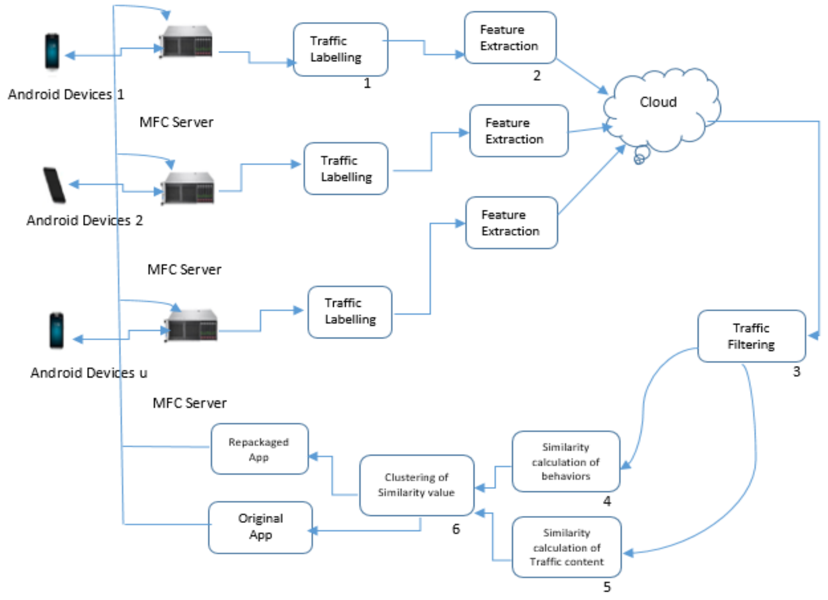


Figure 4: The main procedure of the proposed method for repackaged Android Malware detection

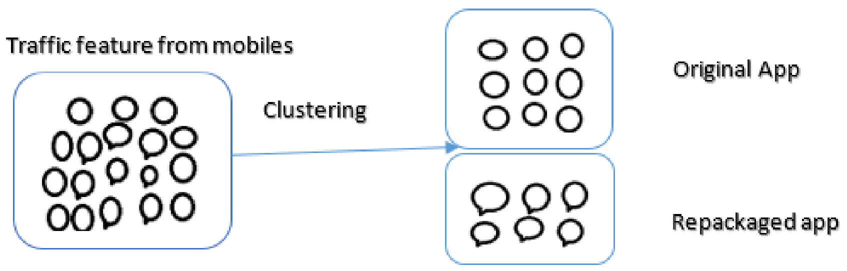
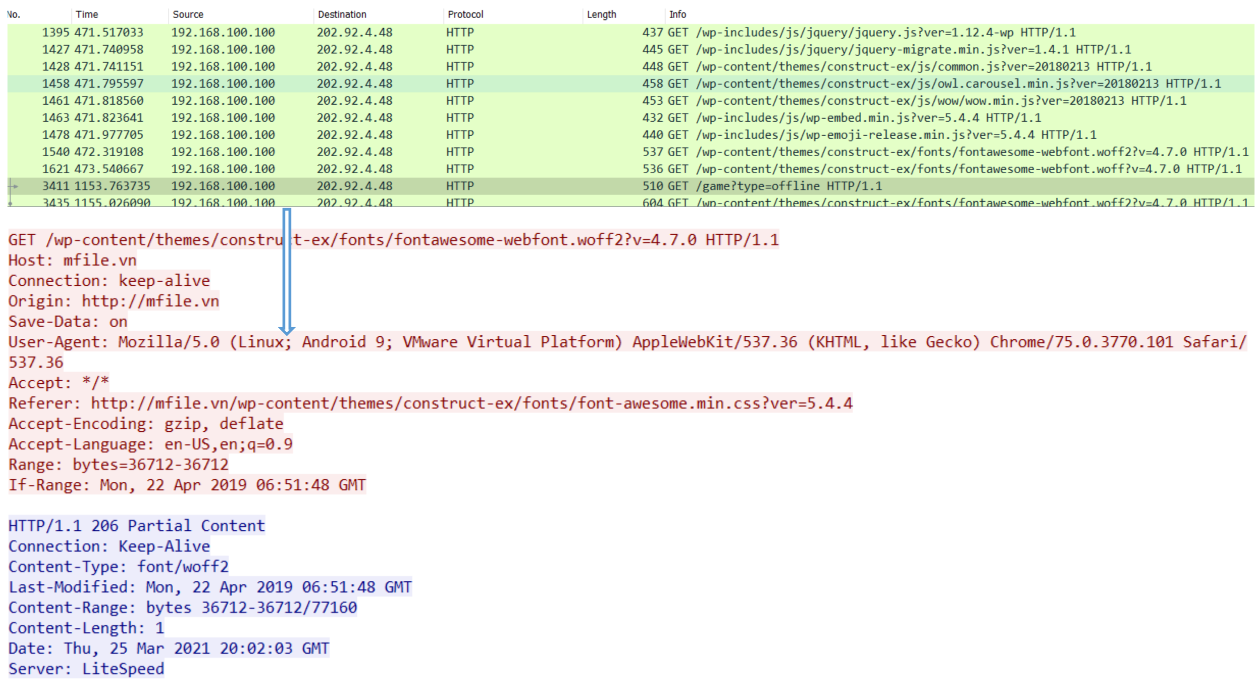


Figure 5: Illustrate the app identification and repackaged malware detection by comparing network traffic.



```
No.   Time         Source           Destination   Protocol   Length  Info
 1395 471.517033   192.168.100.100  202.92.4.48   HTTP       437 GET /wp-includes/js/jquery/jquery.js?ver=1.12.4-wp HTTP/1.1
 1427 471.740958   192.168.100.100  202.92.4.48   HTTP       445 GET /wp-includes/js/jquery/jquery-migrate.min.js?ver=1.4.1 HTTP/1.1
 1428 471.741151   192.168.100.100  202.92.4.48   HTTP       448 GET /wp-content/themes/construct-ex/js/common.js?ver=20180213 HTTP/1.1
 1458 471.795597   192.168.100.100  202.92.4.48   HTTP       458 GET /wp-content/themes/construct-ex/js/owl.carousel.min.js?ver=20180213 HTTP/1.1
 1461 471.818560   192.168.100.100  202.92.4.48   HTTP       453 GET /wp-content/themes/construct-ex/js/wow/wow.min.js?ver=20180213 HTTP/1.1
 1463 471.823641   192.168.100.100  202.92.4.48   HTTP       432 GET /wp-includes/js/wp-embed.min.js?ver=5.4.4 HTTP/1.1
 1478 471.977705   192.168.100.100  202.92.4.48   HTTP       440 GET /wp-includes/js/wp-emoji-release.min.js?ver=5.4.4 HTTP/1.1
 1540 472.319108   192.168.100.100  202.92.4.48   HTTP       537 GET /wp-content/themes/construct-ex/fonts/fontawesome-webfont.woff2?v=4.7.0 HTTP/1.1
 1621 473.540667   192.168.100.100  202.92.4.48   HTTP       536 GET /wp-content/themes/construct-ex/fonts/fontawesome-webfont.woff?v=4.7.0 HTTP/1.1
 3411 1153.763735  192.168.100.100  202.92.4.48   HTTP       510 GET /game?type=offline HTTP/1.1
 3435 1155.026090  192.168.100.100  202.92.4.48   HTTP       604 GET /wp-content/themes/construct-ex/fonts/fontawesome-webfont.woff2?v=4.7.0 HTTP/1.1
```

```
GET /wp-content/themes/construct-ex/fonts/fontawesome-webfont.woff2?v=4.7.0 HTTP/1.1
Host: mfile.vn
Connection: keep-alive
Origin: http://mfile.vn
Save-Data: on
User-Agent: Mozilla/5.0 (Linux; Android 9; VMware Virtual Platform) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/75.0.3770.101 Safari/
537.36
Accept: */*
Referer: http://mfile.vn/wp-content/themes/construct-ex/fonts/font-awesome.min.css?ver=5.4.4
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Range: bytes=36712-36712
If-Range: Mon, 22 Apr 2019 06:51:48 GMT

HTTP/1.1 206 Partial Content
Connection: Keep-Alive
Content-Type: font/woff2
Last-Modified: Mon, 22 Apr 2019 06:51:48 GMT
Content-Range: bytes 36712-36712/77160
Content-Length: 1
Date: Thu, 25 Mar 2021 20:02:03 GMT
Server: LiteSpeed
```

Figure 6: Example of identification of HTTP Traffic flow

Androids app use multiple connections simultaneously; we have a strong correlation network flow generated by the same app [34]. Therefore, exhibit unknown traffics identification feature—namely unknown flows labeled by correlation traffic. The traffic correlation reveals by DNS clustering and retrieves similar traffic. After labeling the network flows, the MFC server extracts the detection feature of labeled traffic.

Feature extraction is divided into two groups, behavior and content feature. Content feature extracted from HTTP Signature and behavior feature extracted from statistics of network traffic flows. In figure 4, in the step of traffic filtering, discard the common traffic and accept uncommon traffic from u traffic set. The common traffic in traffic sets such as port number, server IP address. The common traffic generates the similarity calculation among apps. Similarity can increase the complexity level and exhibit any benefit to detect the repackaged malware.

5.2 Similarity calculation of traffic content

We observe that in Android app used two types of protocol, such as HTTP and HTTPS. HTTP flows only the plaintext content. We used to determine the similarity of different sets produced by the same app. Further precisely, we first reassemble HTTP flows, extract the content of HTTP, storing the packet content of HTTP into a text file as ASCII code.

The similarity is determining by two algorithms such enhanced TF-IDF [35] and cosine similarity. The term frequency deal to calculate the word count in documents. In eq. 1 document denoted as d_i and word count as denoted as w_j and term frequency represents as $TF(w_j, d_i)$.

$$TF(w_j, d_i) = 0.5 + \frac{0.5 \times f(w_j, d_i)}{\max\{f(w_j, d_i) : w \in d_i\}} \qquad (1)$$

where $f(w_j, d_i)$ the number of occurrence of words w_j in the document d_i.

In the eq. 2 to use inverse document frequencies(IDF) determine, amount of information a word carries. Usually, it is applied the logarithm scale function of inverse to determine the words contained in the document.

$$IDF(W_j) = \log\left(\frac{u}{|\{d, w_j \in d\}|}\right) \qquad (2)$$

where u is the number of documents, and $|\{d,\ w_j \in d\}|$ is the number of documents was the exhibit w_j The product of $TF(w_j, d_i)$ and $IDF(w_j)$ determine in Eq.3.

$$TF - IDF\left(w_j, d_i\right) = TF\left(w_j, d_i\right) \times IDF(w_j) \quad (3)$$

After product, all document transforms into the exact size of numeric vectors. Here, we applied the cosine similarity between two document vectors. The two different document vectors, such as $d_i \; and \; d_k$ And the length of the vector is t. The similarity determines between two vectors in eq. 4.

$$CS_{d_{i\,k}} = \frac{\sum_{l=1}^{t} V(d_i)_l \times \sum_{l=1}^{t} V(d_i)_l}{\sqrt{\sum_{l=1}^{t} V(d_i)^2{}_l} \times \sqrt{\sum_{l=1}^{t} V(d_k)^2{}_l}} \quad (4)$$

Finally, for each set d_i compare with remaining sets $u - 1$, and the similarity values would be $CS_{d_{ii}} = 1$ we represent similarity vectors $\{\, CS_{d_{i1,}}, CS_{d_{i1,}}, CS_{d_{i2,}}, CS_{d_{i3,}} \ldots\ldots CS_{d_{iu}} \}$.

Table 2: Statistical feature of network flows to calculate from traffic behavior

SN	Feature	Interpretation
1	DNS-Query (Queries Request)	Count
2	Average-DNS-Interval (Interval between two Queries)	Value
3	Distinct-DNS-Query	Count
4	HTTP-Get /Post Request	Count
5	HTTP-Get/Post-Request-Interval	Value
6	Distinct-HTTP-Get/Post-Request	Count
7	Distinct-Source-Port-Used	Count
8	Send Bytes	Value
9	Received Bytes	Value
10	Send-Average-Length	Value
11	Received-Average-Length	Value
12	Send-Packet	Count
13	Received-Packet	Count
14	Destination	Value
15	TCP-RST	Count
16	No-Fail-SYN	Count

5.3 Similarity calculation of traffic behaviors

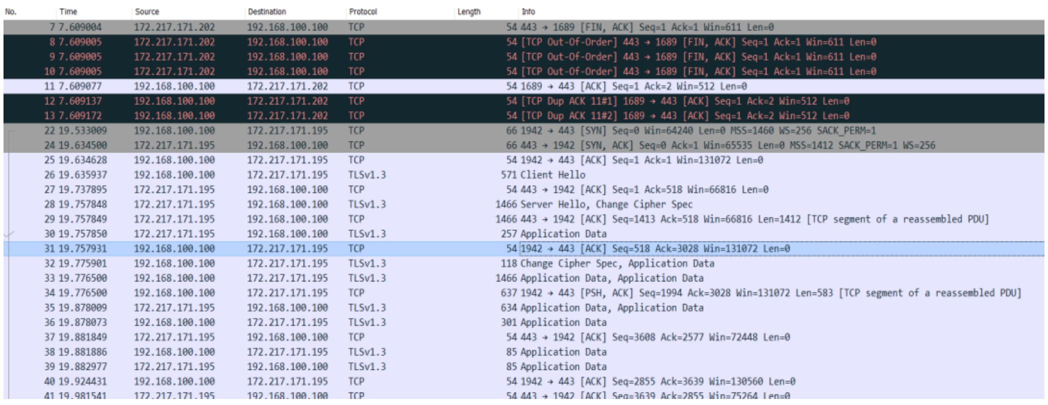
Here, we calculate a similar value of contents that cannot deal with the encrypted flows. Besides, traffic behavior must calculate to deal with the encrypted flows. We use m (m=16) in the statistical analysis study. The statistical feature we selected is given in Table 2. Therefore, the traffic set d_i has a flow vector represented in the eq. 5 for feature extraction. In figure 7 shown the HTTPS traffics flows for encrypted feature vectors.

$$F_{d_i}^{j} = \left(f_j^1, f_j^2, \ldots. f_j^m\right) \quad (5)$$

Where f_j^p denoted the p^{th} statistical feature, $1 \leq p \leq m$. Set d_i can be exhibit in the matrix f_j^p.

$$TB_{n\times m}\,(d_i) = \left(f_{d_i}^1, f_{d_i}^2, \ldots\ldots.. f_{d_i}^n\right)^T \quad (6)$$

where n denotes the number of encrypted flows enclosed in d_i



```
No.   Time        Source           Destination      Protocol  Length  Info
   7  7.609004    172.217.171.202  192.168.100.100  TCP         54 443 → 1689 [FIN, ACK] Seq=1 Ack=1 Win=611 Len=0
   8  7.609005    172.217.171.202  192.168.100.100  TCP         54 [TCP Out-Of-Order] 443 → 1689 [FIN, ACK] Seq=1 Ack=1 Win=611 Len=0
   9  7.609005    172.217.171.202  192.168.100.100  TCP         54 [TCP Out-Of-Order] 443 → 1689 [FIN, ACK] Seq=1 Ack=1 Win=611 Len=0
  10  7.609005    172.217.171.202  192.168.100.100  TCP         54 [TCP Out-Of-Order] 443 → 1689 [FIN, ACK] Seq=1 Ack=1 Win=611 Len=0
  11  7.609077    192.168.100.100  172.217.171.202  TCP         54 1689 → 443 [ACK] Seq=1 Ack=2 Win=512 Len=0
  12  7.609137    192.168.100.100  172.217.171.202  TCP         54 [TCP Dup ACK 11#1] 1689 → 443 [ACK] Seq=1 Ack=2 Win=512 Len=0
  13  7.609172    192.168.100.100  172.217.171.202  TCP         54 [TCP Dup ACK 11#2] 1689 → 443 [ACK] Seq=1 Ack=2 Win=512 Len=0
  22  19.533009   192.168.100.100  172.217.171.195  TCP         66 1942 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
  24  19.634500   172.217.171.195  192.168.100.100  TCP         66 443 → 1942 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1412 SACK_PERM=1 WS=256
  25  19.634628   192.168.100.100  172.217.171.195  TCP         54 1942 → 443 [ACK] Seq=1 Ack=1 Win=131072 Len=0
  26  19.635937   192.168.100.100  172.217.171.195  TLSv1.3    571 Client Hello
  27  19.737895   172.217.171.195  192.168.100.100  TCP         54 443 → 1942 [ACK] Seq=1 Ack=518 Win=66816 Len=0
  28  19.757848   172.217.171.195  192.168.100.100  TLSv1.3   1466 Server Hello, Change Cipher Spec
  29  19.757849   172.217.171.195  192.168.100.100  TCP       1466 443 → 1942 [ACK] Seq=1413 Ack=518 Win=66816 Len=1412 [TCP segment of a reassembled PDU]
  30  19.757850   172.217.171.195  192.168.100.100  TLSv1.3    257 Application Data
  31  19.757931   192.168.100.100  172.217.171.195  TCP         54 1942 → 443 [ACK] Seq=518 Ack=3028 Win=131072 Len=0
  32  19.775901   192.168.100.100  172.217.171.195  TLSv1.3    118 Change Cipher Spec, Application Data
  33  19.776500   192.168.100.100  172.217.171.195  TLSv1.3   1466 Application Data, Application Data
  34  19.776500   192.168.100.100  172.217.171.195  TCP        637 1942 → 443 [PSH, ACK] Seq=1994 Ack=3028 Win=131072 Len=583 [TCP segment of a reassembled PDU]
  35  19.878009   172.217.171.195  192.168.100.100  TLSv1.3    634 Application Data, Application Data
  36  19.878073   192.168.100.100  172.217.171.195  TLSv1.3    301 Application Data
  37  19.881849   172.217.171.195  192.168.100.100  TCP         54 443 → 1942 [ACK] Seq=3608 Ack=2577 Win=72448 Len=0
  38  19.881886   192.168.100.100  172.217.171.195  TLSv1.3     85 Application Data
  39  19.882977   172.217.171.195  192.168.100.100  TLSv1.3     85 Application Data
  40  19.924431   192.168.100.100  172.217.171.195  TCP         54 1942 → 443 [ACK] Seq=2855 Ack=3639 Win=130560 Len=0
  41  19.981541   172.217.171.195  192.168.100.100  TCP         54 443 → 1942 [ACK] Seq=3639 Ack=2855 Win=75264 Len=0
```

Figure 7: Encrypted traffic flows for behavior feature vectors.

In eq. 7, illustrate a feature set, but that is not identical in size in encrypted flows. The behavior of $TB_{n_i \times m}\,(d_i)$ and $TB_{n_k \times m}\,(d_k)$ are not identical in terms of dimensions. We determine the Frobenius norm for $TB_{n_i \times m}\,(d_i)$ and $TB_{n_k \times m}\,(d_k)$. The procedure is presented in eq. 7, where f_j^1 normalized the value into[0,1]:

$$FN(d) = \sqrt{\sum_{j=1}^{n} \sum_{p=1}^{m} normalize \left| f_j^p \right|^2} \tag{7}$$

After that, calculate the distance between $TB_{n_i \times m}\,(d_i)$ and $TB_{n_k \times m}\,(d_k)$ is determine into eq. 8.

$$dis\big(d_i, d_j\big) = |FN(d_i) - FN(d_k)| \tag{8}$$

Finally, determine the similarity between $TB_{n_i \times m}\,(d_i)$ and $TB_{n_k \times m}\,(d_k)$ in Eq. 9. In the below process n_i and n_k Not more than 1.

$$BS_{d_{ik}} = 1 - \frac{dis\big(d_i, d_j\big)}{\max\,[\,dis\big(d_i, d_j\big), r, q = 1,2,3 \ldots u, r \neq q]} \tag{9}$$

$BS_{d_{ik}}$ Set to 1 if coming to the same behavior and otherwise set to 0. Similarity

5.4 Clustering of similarity values and repackaged malware detection

The similarity of traffic content $CS_{d_{ik}}$ and feature extraction (behavior) similarity $BS_{d_{ik}}$, we determine the final similarity $S_{d_{ik}}$ among the d_i *and* d_k. Illustrate in Eq. 10. The set d_i has a vector length $S_{d_i} = (S_{d_{i1}}, S_{d_{i2}}, S_{d_{i3}}\ \ldots\ S_{d_{iu}})$. Here $\{S_{d_i}\}$, $i = 1,2,3, \ldots u$ clustering and detect the repackaged app.

$$S_{d_{ik}} = q \times CS_{d_{ik}} + (1 - q) \times BS_{d_{ik}} \tag{10}$$

Eq. 10 used the density peak clustering approach [36]. In this approach, we have a high-density cluster in the center from its neighbor. It keeps a significant distance from the density cluster. The density Peak clustering approach automatically recognizes the correct cluster. This approach is suitable for detect repackaging malware because we don't know whether the app is repackaged or not and how many types of repackaging malware. Density peak clustering is the optimum solution to this kind of problem. Clustering, we labeled the original app and repackaged the app. We predict that most android devices run the original app. Therefore, in the clustering process, the most significant area of Voxels in the cluster is recognized as the original and continuing cluster viewed as the suspicious app. The suspicious app may be repackaged malware or affected by other malware. However, adware would be repackaged malware. Original can have recognized as a suspected app due to different behaviors of the app. We minimized the false alarm to verify the hostname name of the server. The repackaged recognize into VirusTotal by hostname of the server in the cluster. The original app, if all hostname names of the server, was found clean in the cluster and expected behavior, otherwise recognized as repackaged malware.

7. EVOLUTION

We downloaded the 400 original and 400 similar repackaged apps from Andozoo to run the experiment. All downloaded 400 apps checked by Virustotal. We found the 296 apps infected by malware verified to Virustotal, shown the result of 20 apps in Table 1. Nevertheless, we obtained a more realistic performance by our experiment. We set up 8 Android 9.0 OS on VMware 16.x Pro under win 10, all downloaded app installed on different android devices. We collected all network traffic using TCPDump. The fog Server (assume win-10 machine) used the TCPDump to collect all network traffic. Here we saved the pcap files. To extract the feature, we used the splitcap to reestablish TCP flow from pcap files. To find the extracted traffic feature from HTTP used the Wireshark. The evaluate of $CS_{d_{ik}}$and $BS_{d_{ik}}$ Used the scikit-learn and NumPy. The clustering code (cluster-peak_density) was downloaded from Github.

$$Accuracy = \frac{TP+TN}{TP+FN+FP+TN} \tag{11}$$

$$F - Measure = \frac{2*Precesion*recall}{recall+Precesion} \tag{12}$$

$$Precesion = \frac{TP}{TP+FP} \tag{13}$$

$$Recall = \frac{TP}{TP+FN} \tag{14}$$

In previous works [8,14] to evaluated the performance proposed approach. We have measured the accuracy and F-measure for each app, define into eq. 11 and eq. 12. In eq. 12, evaluated by from $Recall\,(\text{eq}.\,13)$ and $Precesion(\text{eq}.\,14)$. We have calculated the F-measure and accuracy for each repackaged app. Table 3 and figure 8 represent the HTTP and HTTPS(Encrypted) traffic flows.

Table3: Detail of description of the malicious apps with encryption and without encryption traffic flows

S.N.	AndroZoo APK	App Types	# of HTTP Flows	# of HTTPS Flows
1	939b53399a59b34d839dc055682db549.APK	Game	353	151
2	092b3663cb136f5192bfa7e1dc147420.APK	Browser	1641	1368
3	093db22b159645d0d7201d9048f0541b.APK	Chat	767	1345
4	194adfa70978e2828a9397204b33bc51.APK	Game	231	35
5	220c905d9f1286285356c9511be9d581.APK	Game	502	398
6	271e7feb283bd249755be37c7ebf717a.APK	Game	678	233
7	293d5e8a0faf770cd00367e6e84ce2a8.APK	Media Player	1226	634
8	349f4cf8ce6ab7652da35a89aaa10b3e.APK	Music	227	1012
9	508e82e3ec3944ce257f126363d4ea7d.APK	Game	172	381
10	698b81d63841362c35be0543870b4a5d.APK	News	763	990
11	718e60ac1ad4bd6e9e3ee46994ac8d68.APK	News and Weather	1108	548
12	814e01e693bea00db481dc30be5aeb1f.APK	Fitness	1098	926
13	916a1677ec6fd9798a82381d921a5b9b.APK	Online learning	2391	1443
14	1317f0080e1b5d0a55876116e1dfd22b.APK	Game	105	109
15	642ec177fa7db0bb206b1fd2746798db.APK	Email	243	268
16	98f4b2087b408ee174fd07f7898b9127e.APK	Chat	280	239
17	87b179e18c1947e365a7c9db4f0a3052.APK	Game	1942	810
18	87b6994c27f474b9874f292ee27bda76.APK	Game	622	253
19	408f5eca5cd8be5623fe2fcbfb84d093.APK	Chat	2468	724
20	986ad626e433d8c7f9533fd33540ef1b.APK	Game	2250	1288

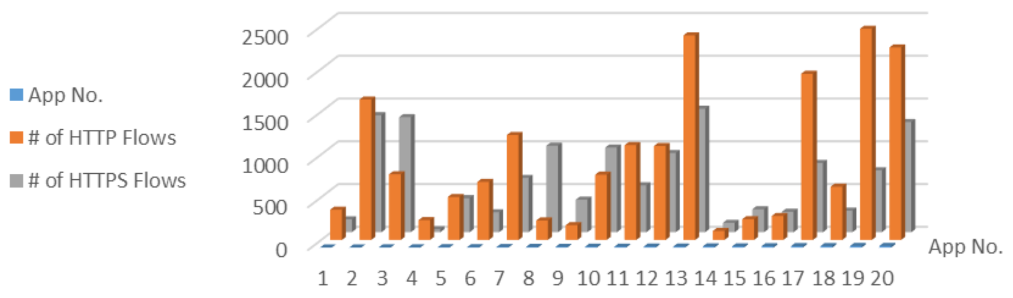


Figure 8: Network-based encrypted and unencrypted flows of 20 apps

Table 4: Dataset for Malicious android app detection

	Number of apps	**Number of Execution**	**Number of HTTP Flows**	**Number of HTTPS Flows**
Original app	400	50	705780	410000
Repackaged App	400	20	315000	198000

Table 5: Detection of unknown app accuracies with different features

S. N.	**Accuracy(in %) With 16 Features**	**Accuracy(in %) with 11 features**
1	80	80
2	92.6	92.6
3	82.7	81.5
4	63	62.3
5	98.9	97.3
6	99	98
7	85.3	84.3
8	96.9	88.5
9	99	98
10	98	98
11	99	98.2
12	98	98
13	99.5	98.7
14	70	70
15	98.8	98.4
16	99	99
17	66.7	66.5
18	78.8	78.6
19	98.4	98
20	89.3	87.8
Average	**89.64**	**88.68**

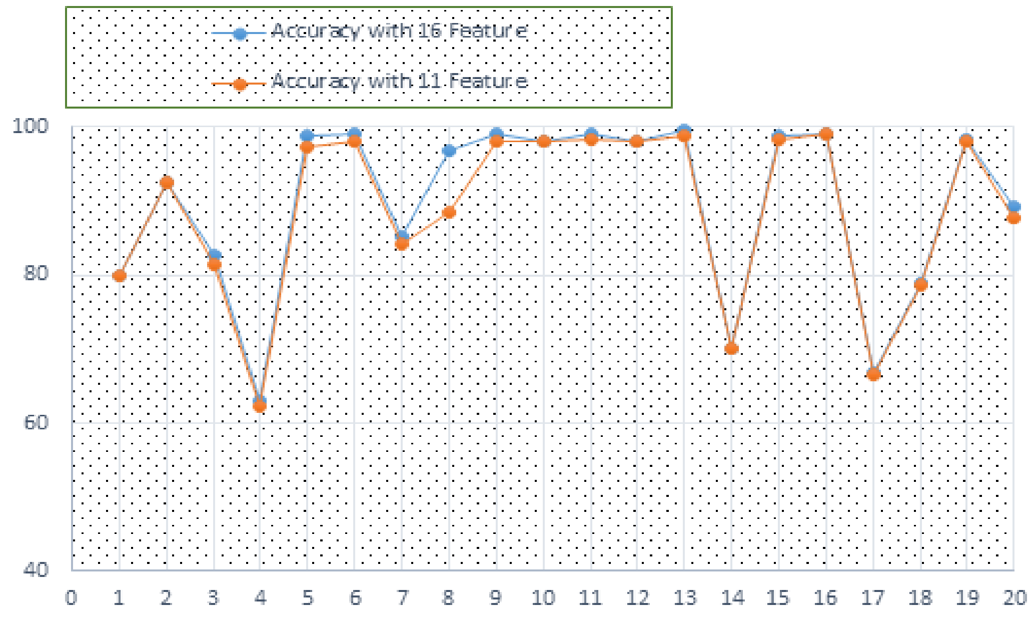


Figure 9: Comparison of accuracy with different extraction feature of 20 apps

We have designed F-measure and accuracy of original apps and repackaged apps. In the experiment, we took 50 original app traffic sets and 20 repackaged app traffic sets in Table 4, specifically TP+FN=20 and TN+FP=50 for every app. In VirusTotal detected, 296 apps are affected by malware, and 104 apps are safe. In this calculation, TP+FN=104*50=5200 and TN+FP=104*20=2080 for repackaged safely. For malware app TP+FN=296*50 =14800 and 296 *20=5920.This is for in our tests. In VirusTotal, 296 apps are recognized as malware app, and 104 apps are original apps.

In figure 9 and Table 5 depicted that app number {5,6,9,10,11,13,15,16 and 19} have more than 98.4% accuracy with 16 feature extraction point but with 11 feature extraction have less accuracy. However, more feature extraction increased the complexity of the detection of the approach. The low number of the feature has very low complexity. Table 6 depicted the average accuracy and F-Measure with 16 features and 11 feature extraction points to detect the repackaged malware apps and original apps. We found a 97.7% good accuracy rate with 16 feature extraction points with high complexity measures and 96.9% accuracy with 11 feature extraction points with less complexity than 16 feature extraction points.

We found that more feature extraction increased more computational time and increased the accuracy of detecting repackaged malware. Less feature extraction decreased the computation time and decreased accuracy of detection of the repackaged malware apps.

Table 6: Average accuracy and F-Measure detection with different feature extraction

	Average Accuracy	**Average F-Measure**
Detection with 16 features extraction Parameter	97.7	0.96
Detection with 11 features extraction Parameter	96.9	0.94

8. CONCLUSION

In this paper, we enhanced proposed an approach to the detection of Android repackaged malware. Experimental works used Fog commuting to reduce the end device (Mobile node) load for Android malware detection. In the current scenario, most of the techniques used static analysis to detect the repackaged apps. Here we used the dynamic approach for detection and repackaged malware with analysis of network traffics. We used the cluster-peak_density and feature extraction of network traffics of each app to detect the app with high accuracy of 97.7%. We found that if we increased more feature extraction parameters for detection, malware apps also increased the CPU consumption, and low feature extraction parameters decreased the accuracy also decreased the CPU consumption. In the future, we try to improve the accuracy and decrease the CPU consumption.

REFERENCES:

[1] Statista, Mobile Operating Systems' Market Share Worldwide from January 2012 to December 2019, Statista, Hamburg, Germany, 2009, https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systemssince-2009/.

[2] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A survey of app store analysis for software engineering," IEEE Transactions on Software Engineering, vol. 43, no. 9, pp. 817–847, 2017.

[3] https://www.appbrain.com/stats/number-of-android-apps,2020.

[4] L. Li, D. Li, T. F. Bissyande et al., "Understanding android app piggybacking: a systematic study of malicious code grafting," IEEE Transactions on Information Forensics and Security, vol. 12, no. 6, pp. 1269–1284, 2017.

[5] M. Fan, J. Liu, X. Luo et al., "Android malware familial classification and representative sample selection via frequent subgraph analysis," IEEE Transactions on Information Forensics and Security, vol. 13, no. 8, pp. 1890–1905, 2018.

[6] J. Zhang, Z. Qin, K. Zhang, H. Yin, and J. Zou, "Dalvik opcode graph-based android malware variants detection using global topology features," IEEE Access, vol. 6, pp. 51964–51974, 2018.

[7] S. Garg, S. K. Peddoju, and A. K. Sarje, "Network-based detection of android malicious apps," International Journal of Information Security, vol. 16, no. 4, pp. 385–400, 2016.

[8] A. Shabtai, L. Tenenboim-Chekina, D. Mimran, L. Rokach, B. Shapira, and Y. Elovici, "Mobile malware detection through analysis of deviations in application network behavior," Computers & Security, vol. 43, pp. 1–18, 2014.

[9] L. Xie, X. Zhang, J.-P. Seifert, and S. Zhu, "pBMDS: a behavior-based malware detection system for cellphone devices," in Proceedings of the ird ACM Conference on Wireless Network Security, ACM, New York, NY, USA,pp. 37–48, 2010.

[10] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in Proceedings of the 1st ACM Workshop on Security and Privacy in Smart-phones and Mobile Devices, ACM, Chicago, IL, USA,pp. 15–26, October 2011.

[11] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, "MADAM: a multi-level anomaly detector for android malware," in Proceedings of the International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security, vol. 12, Springer, St. Petersburg, Russia, pp. 240–253, 2012.

[12] Y. Zhang, M. Yang, B. Xu et al., "Vetting undesirable behaviors in android apps with permission use analysis," in Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, ACM, Berlin, Germany, pp. 611–622, November 2013.

[13] S. Wang, Z. Chen, X. Li, L. Wang, K. Ji, and C. Zhao, "Android malware clustering analysis on network-level behavior," in Proceedings of the International Conference on Intelligent Computing, Springer, Liverpool, UK, pp. 796–807, August 2017.

[14] G. He, B. Xu, and H. Zhu, "AppFA: a novel approach to detect malicious android applications on the network," in Security and Communication Networks, Wiley, Hoboken, NJ, USA, 2018.

[15] M. Sun, X. Li, J. C. S. Lui, R. T. B. Ma, and Z. Liang, "Monet: a user-oriented behavior-based malware variants detection system for android," IEEE Transactions on Information Forensics and Security, vol. 12, no. 5, pp. 1103–1112, 2017.

[16] Y. Zhou and X. Jiang, "Dissecting android malware: characterization and evolution," in Proceedings of the 2012 IEEE Symposium on Security and Privacy, IEEE, San Francisco, CA, USA, pp. 95–109, 2012.

[17] N. W. Lo, S. K. Lu, and Y. H. Chuang, "A framework for third-party android marketplaces to identify repackaged apps," in Proceedings of the 2016 IEEE 14th International Conference on Dependable, Autonomic and Secure Computing, 14th International Conference on Pervasive Intelligence and

Computing, 2nd International Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress, pp. 475–482, Auckland, New Zealand, August 2016.

[18] L. Li, J. Gao, M. Hurier et al., “Androzoo++: collecting millions of android apps and their metadata for the research community,” 2017, https://arxiv.org/abs/1709.05281.

[19] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge computing: vision and challenges,” IEEE Internet of ings Journal, vol. 3, no. 5, pp. 637–646, 2016.

[20] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, “Detecting repackaged smartphone applications in third-party android marketplaces,” in Proceedings of the Second ACM Conference on Data and Application Security and Privacy, ACM, Antonio, TX, USA, pp. 317–326, February 2012.

[21] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, “Towards a scalable resource-driven approach for detecting repackaged android applications,” in Proceedings of the 30th Annual Computer Security Applications Conference, ACM, New Orleans, LA, USA, pp. 56–65, December 2014.

[22] K. Chen, P. Wang, Y. Lee et al., “Finding unknown malice in 10 seconds: mass vetting for new threats at the google-play scale,” in Proceedings of the USENIX Security Symposium, vol. 15, Washington, DC, USA, August 2015.

[23] Z. Wang, C. Li, Y. Guan, and Y. Xue, “Droidchain: a novel malware detection method for android based on behavior chain,” in Proceedings of the 2015 IEEE Conference on Communications and Network Security (CNS), IEEE, Florence, Italy, pp. 727-728, September 2015.

[24] K. Tian, D. D. Yao, B. G. Ryder, G. Tan, and G. Peng, “Detection of repackaged android malware with code-heterogeneity features,” IEEE Transactions on Dependable and Secure Computing, vol. 17, no. 1, pp. 64–77, 2017.

[25] A. De Lorenzo, F. Martinelli, E. Medvet, F. Mercaldo, and A. Santone, “Visualizing the outcome of dynamic analysis of android malware with vizmal,” Journal of Information Security and Applications, vol. 50, Article ID 102423, 2020.

[26] M. Lin, D. Zhang, X. Su, and T. Yu, “Effective and scalable repackaged application detection based on user interface,” in Proceedings of the 2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), IEEE, Atlanta, GA, USA, May 2017.

[27] G. Meng, M. Patrick, Y. Xue, Y. Liu, and J. Zhang, “Securing android app markets via modelling and predicting malware spread between markets,” IEEE Transactions on Information Forensics and Security, vol. 14, no. 7, pp. 1944–1959, 2018.

[28] X. Wu, D. Zhang, X. Su, and W. Li, “Detect repackaged Android application based on HTTP traffic similarity http traffic similarity,” Security and Communication Networks, vol. 8, no. 13, pp. 2257–2266, 2015.

[29] J. Malik and R. Kaushal, “Credroid: android malware detection by network traffic analysis,” in Proceedings of the 1st ACM Workshop on Privacy-Aware Mobile Computing, ACM, Paderborn, Germany, pp. 28–36, July 2016. 18 Security and Communication Networks

[30] A. Zulkifli, I. R. A. Hamid, W. M. Shah, and Z. Abdullah, “Android malware detection based on network traffic using decision tree algorithm,” in Proceedings of the International Conference on Soft Computing and Data Mining, Springer, Senai, Malaysia, pp. 485–494, January 2018.

[31] Z. Chen, Q. Yan, H. Han et al., “Machine learning based mobile malware detection using highly imbalanced network traffic,” Information Sciences, vol. 433-434, pp. 346–364, 2018.

[32] G. He, B. Xu, L. Zhang, and H. Zhu, “Mobile app identification for encrypted network flows by traffic correlation,” International Journal of Distributed Sensor Networks, vol. 14, no. 12, pp. 1–17, 2018.

[33] Q. Xu, Y. Liao, S. Miskovic, Z.M. Mao, M. Baldi, A. Nucci, et al., "Automatic generation of mobile app signatures from traffic observations", 2015 IEEE Conference on Computer Communications (INFOCOM), pp. 1421-1489, 201

[34] G. He, B. Xu and H. Zhu, "Identifying mobile applications for encrypted network traffic", 2017 Fifth International Conference on Advanced Cloud and Big Data (CBD), pp. 279-284, 2017.

[35] L. Li and S. Qu, “Short text classification based on improved ITC,” Journal of Computer and Communications, vol. 1, no. 4, pp. 22–27, 2013

[36] A. Rodriguez and A. Laio, “Clustering by fast search and find of density peaks,” Science, vol. 344, no. 6191, pp. 1492–1496, 2014.